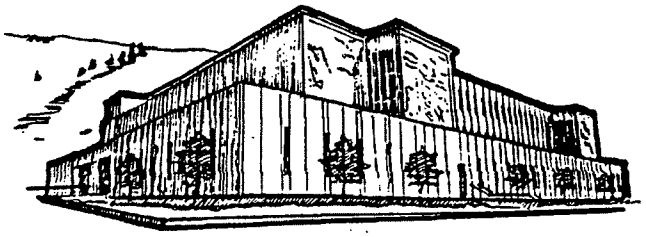
1993

Client/server prototype of visual price-matching system for on-line stock transactions

Hong Fan
The University of Montana



A Client/Server Prototype of Visual Price-Matching System for On-Line Stock Transactions

By

Hong Fan

B.S. Nanking Institute of Posts and Telecommunications

Radio Communications

Presented in partial fulfillment of the requirements

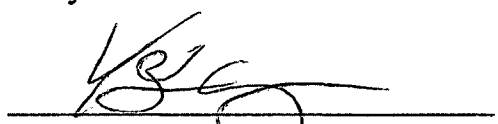
for the degree of

Master of Science

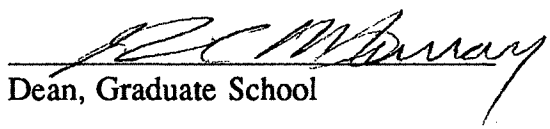
University of Montana

1993

Approved by



Chairman, Board of Examiners



Dean, Graduate School

February 9, 1993

Date

3-23-93

TABLE OF CONTENTS

Acknowledgements I
Table of Contents II
List of Figures III
Chapter 1 Introduction 1
1.1 Background 1
1.2 Problem Statement 1
1.3 Project Scope 2
1.4 Facility/Compatibility 2
Chapter 2 System Analysis and Design 3
2.1 System Layout 3
2.2 System Server Internal Structure 5
2.3 System Client User Interface 7
2.4 System Simulation 8
2.5 Computerization of Price Matching 8
Chapter 3 System Implementation 10
3.1 Implementation Approach 10
3.2 Concurrent Server 10
3.3 Connection-Oriented Access 11
3.4 Single-process or Multi-process Server 12
3.5 Concurrent Server Algorithm 13
3.5.1 The Generic Concurrent Server Algorithm 13
3.5.2 Process Structure 14
3.6 A Modified Concurrent Server and Algorithm 15
3.6.1 Server Service Requirement 15
3.6.2 System Server Structure 16
3.7 System Client 22
3.8 The Socket Interface 24
3.8.1 The Socket Call 24
3.8.2 The Connect Call 24
3.8.3 The Write Call 24
3.8.4 The Read Call 25
3.8.5 The Close Call 25
3.8.6 The Bind Call 26
3.8.7 The Listen Call 26
3.8.8 The Accept Call 27
3.8.9 Using Socket Calls in a Program 27
Chapter 4 Client System User Interface 29
4.1 Why MOTIF 29
4.2 Motif Widget Overview 29
4.3 User Interface Design 32
4.4 Handling Other Input Sources 39
Chapter 5 Price Matching 41
5.1 Order's Data Structure 41
5.2 Price Matching Algorithm 45
Chapter 6 Conclusion 58

List of Figures

Figure2.1 3
Figure2.2 4
Figure2.3 5
Figure 3.1 15
Figure 3.2 17
Figure 3.3 17
Figure 3.4 19
Figure 3.5 20
Figure 3.6 28
Figure 4.1 30
Figure 5.1 55
Figure 5.2 55
Figure 5.3 56
Figure 5.4 56
Figure 5.5 57

ACKNOWLEDGEMENTS

I would particularly like to thank Dr. Youlu Zheng for his effort and support, and the initial idea for this project. His expertise was the technical backbone of this paper. I am proud to be his student.

I also would like to thank Professor Alden Wright and Professor James Lowes for their invaluable guidance, and to Dr. Ray Ford for providing his personal workstation and the software in the earliest stage of the project.

CHAPTER 1 INTRODUCTION

1.1 Background

Computer networks in the stock market are currently used extensively for the delivery of stock orders. Customers and brokers, in order to know the current market trading state and the good time to buy and sell stock, must spend much time accessing a variety of data, such as the stock market average, market indicators, and stock quotations from various sources. Although these data are important and helpful, they are static and historical. They are also tedious to read and understand. Brokers could make a more accurate and quicker judgement on each transaction if the information, such as the current pending state of asked-bid orders of certain stocks, was visually available. Price matching, which is a time-consuming, exhausting, and high-pressure procedure, is accomplished by specialists at the various stock exchanges. The computerization of price matching would therefore greatly improve the efficiency of stock exchanges.

1.2 Problem Statement

The objective of this project is to build a prototype for a visual price-matching system that would computerize the price matching and provide dynamic scenarios with histogram information on different stocks to brokers in a small scale network environment.

1.3 Project Scope

In this project, a client/server model network is developed using MOTIF[1] to simulate some common stock transactions. Also, a prototype algorithm is designed to simulate the stock price-matching. The project focuses on the design and implementation of the client/server model in a small network environment. Although the algorithm is designed to simulate the stock price matching, further development is needed to match the requirements of a real stock transaction system. The evaluation of the system performance has not been done. The involvement of machines from an off-campus network is highly recommended to test the system performance.

1.4 Facility/Compatibility

The platform for this project will be a network of IBM RISC/6000 workstations running AIX 3.2 and X11 R4. AIX 3 merges several operating system functions and interfaces. There are many differences between AIX 3 and 4.3 BSD as a result of the conformance of AIX 3 to the POSIX and ANSI C standards. Some typical conflicts are in the functions of random number generation and the TCP/IP socket API implementation.

---

[1]. A user interface library based on X Windows from Open Software Foundation.

CHAPTER 2 SYSTEM ANALYSIS AND DESIGN

2.1 System Layout

The model is designed to simulate a networked stock trading system. It would allow brokers to obtain the trade information they are interested in and to accomplish trade operations through networks. Brokers could see into the marketplace by studying the activities reported on their computer screens, monitoring trading states of different stocks at the same time, and buying or selling stock as their customers desire through networks. Figure 2.1 shows the layout of the network system:

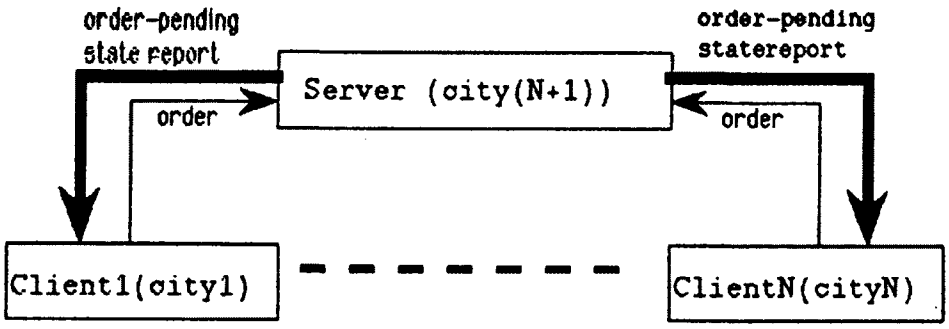


Figure 2.1 System layout of the client/server model of visual-matching system for on-line stock transactions

A real-world implementation of the system would consist of one server site and many client sites, which could be located in different cities. The server system would be located in the stock trading center. It would be mainly responsible for all the stock price-matchings and for providing stock trading state information upon clients' requests. The

client systems would be located where brokers work. From any of the client sites, brokers could send order requests to the server by means of a network. However, the most important feature of the system is that brokers could retrieve very useful information from the server, such as the stock asked-bid order-pending state.

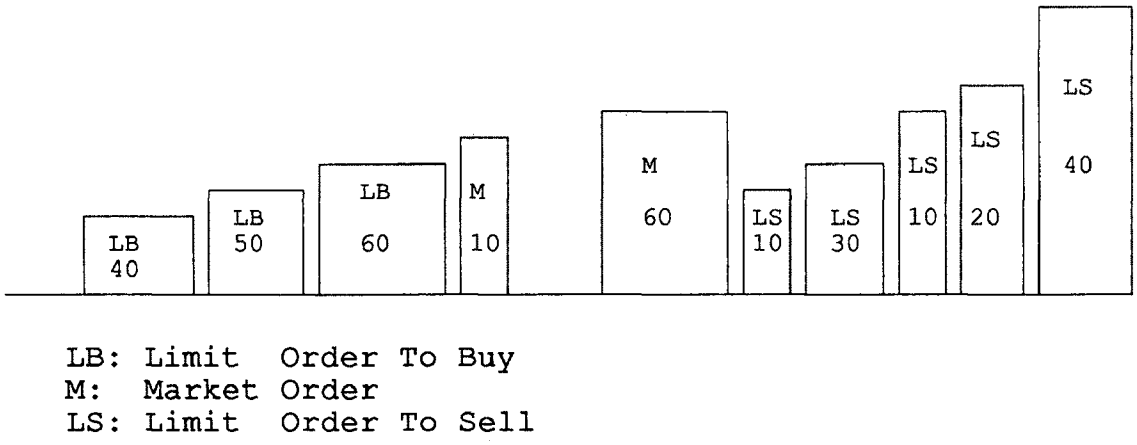


Figure 2.2 A sample IBM stock asked-bid order-pending state histogram.

Figure 2.2 shows a sample IBM stock asked-bid order-pending state histogram that could be obtained by sending a request to the server through the network. The width and height of each rectangle stand for an order's volume and price respectively. The rectangles on the left are bid orders; those on the right are asked orders. The information in this format lets brokers grasp the current trading state immediately and have a more comprehensive picture of the current stock trading state at anytime. This kind of histogram is not available under the current stock trading computer system.

2.2 System Server Internal Structure

This section describes the actual system implemented in this project.

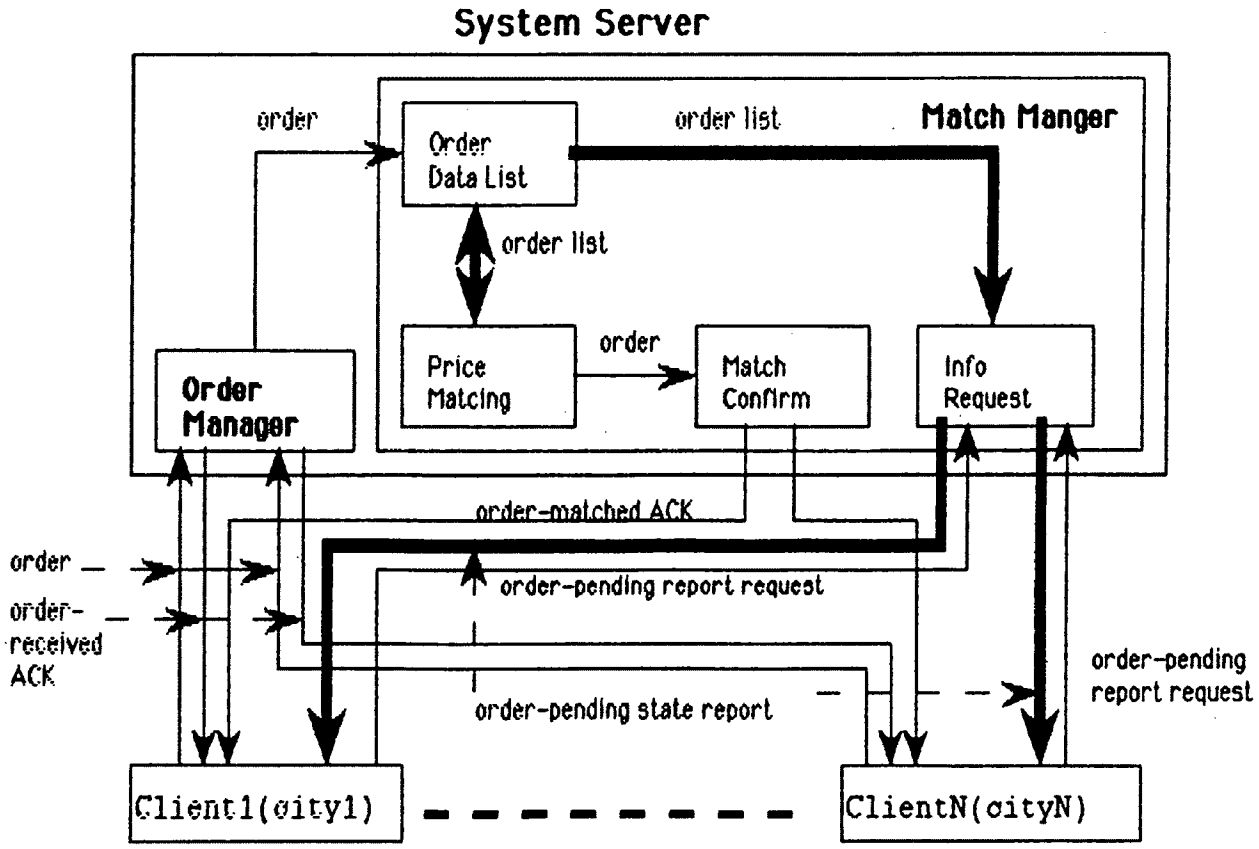


Figure 2.3 Internal layout of the client/server model of visual-matching system for on-line stock transactions

Figure 2.3 shows the internal structure of the system. The primary data entity of the system is the "order", which is represented in the form of entry data dictionary.

```
order       = ask_or_bid + stockname + volume + price +
              type + none_or_all
ask_or_bid = [ "ask_order" | "bid_order" ]
```

```
stockname  = ["HarBraceJ" | "HarBraceJpf" | "Harland" |
             "HarleyDav" | "HarmanInt"   | "Harnisch"|
             "Harris"    | "Harsco"      |"Hartmarx" |
             "HattersSec" ][2]
volume     = { 0-9 }
price      = { 0-9 } + . + { 0-9 }
type       = { "limit order to buy"  |
             "limit order to sell" |
             "market order" }
none_or_all  = [ "yes" | "no" ]
```

Another important data entity is the stock order-pending list as described in section 2.1, which could be obtained by sending info-requests from clients.

```
order_list   = { order }
info-request = stockname
```

In addition, two other data entities, order-received acknowledgement and order-matched acknowledgement, are used for confirmation to data communications.

```
order-received_ack = received + order
received           = [ "received" ]
order-matched_ack  = matched + order + matchprice
matchprice         = price
```

The system server is made up of two main managers: a Match Manager and an Order Manager. The Order Manager is responsible for collecting orders from clients and sending them to the Match Manager. Upon the receipt of a new coming order, the Order

[2] In this project, the above ten stocks are selected to simulate on-line stock transactions.

Manager replies to the corresponding client by sending an order-received acknowledgement, indicating the stock order has been received successfully. The Order Manager then sends the new order to the "order data list" of the Match Manager, where all current pending stock orders are stored. The Match Manager consists of "order data list", "Price Matching", "Match Confirm", and "Info Request". For each stock, there are two lists for bid orders and asked orders in the "data order list". "Price matching" accesses orders from the order data list, performs the price matching, and updates the order data list. When an order is matched, "Match Confirm" sends an order-matched confirmation to the sending client, indicating that the stock order has been matched successfully. Also, related information, stock volume and stock price, will be sent back to the sending client as book keeping. Upon the client's information request, "Info request" searches for the desired stock list, retrieves order data, and sends snapshots of the current asked-bid order-pending state report to the requesting client.

2.3 System Client User Interface

In this project, all clients are within the campus network environment, but represent clients located in different cities in a real world system. The client system has a user interface including a menu bar with several pulldown menus, a stock list field, a stock information field, and a message board. All the current stock's symbol, price, highest price, lowest price, percent yield, and PE will be displayed in the stock list field. Selection of a stock causes a pop-up window to appear, with the corresponding asked-bid order-pending state histogram in it. The commands in the pulldown menu from the menu

bar allow brokers to make normal stock transaction operations such as buy or sell while brokers are monitoring certain stock's asked-bid order-pending state histogram. Brokers also can watch trading activities in different stocks simultaneously. The information field displays some related stock market indicators such as the Standard & Poor's 500, the NASDAQ National Market System Composite Index, the NYSE Composite Index, and the AMEX Market Value Index. In this project, these are dummy values that do not reflect current situation. The message board shows the current transaction or operation status messages such as order-received acknowledgement and order-matched acknowledgement.

2.4 System Simulation

To make the simulation as realistic as possible, the data of each order such as order price, order volume, and order type, are generated under a "desired" probability distribution. By increasing the data sending rate from client sites, we can simulate stock trading and examine how well the server processes price matching. Brokers might manually input the expected stock order information (price, volume, and type, etc.) by using a mouse point and clicking the commands under the top menu bar. Thus they could monitor how the order is matched on the stock pending state histogram.

2.5 Computerization of Price Matching

Another feature of the system is the computerization of price matching. There are different kinds of stock orders such as market price order, limit order to buy, limit order

to sell. (See Appendix for detail) Different orders have different customer transaction requirements in terms of price and volume. The determination of each stock transaction can be easily affected by human factors. To facilitate the simulation, an average of the asked price and the bid price is used for the computerization of price match when a market price bid order and a market price asked order are matched at this phase. To make the computerization trading more feasible in the future, human factors should be treated more realistically.

CHAPTER 3 SYSTEM IMPLEMENTATION

3.1 Implementation Approach

The target system is designed to allow the information to be exchanged through Internet with client sites located anywhere there are available Internet connections. This prototype network system design is based on the Berkeley socket mechanism. Although the SUN RPC or the System V's Transport Layer Interface could have been used, the Berkeley socket mechanism appeared to be a more mature approach to implementing the client/server model, especially in the popular Internet environment.

3.2 Concurrent Server

Although an iterative server, which processes one request at a time, is simpler and easier to build, this model uses a concurrent server, which handles multiple requests at the same time. The concurrent server implementation also yields better performance.

Because a stock transaction requires substantial communications, an iterative server is limited to one client at a time, so once a client contacts the server, the server must refuse, block, or ignore subsequent requests until the first user finishes. Clearly, such a design limits the utilization of the server, and prevents more than one client from accessing the server at any given time.

On the other hand, concurrency improves response time if:

- forming a response requires significant I/O,
- the processing time required varies dramatically among requests, or
- the server executes on a computer with multiple processors.

The concurrent stock price-matching server can accept multiple orders and information requests from clients and can handle many I/O events during each stock transaction. Because system performance is a critical issue in stock transaction design, the concurrent approach is more feasible than the iterative approach.

3.3 Connection-Oriented Access

Connectivity depends on the transport protocol that a client uses to access a server. This model uses the TCP protocol suite, which provides a connection-oriented transportation service. This is preferred over the UDP protocol which provides a connectionless service in which messages can be lost, duplicated, or arrive out of order. A connection-oriented server is easier to program and with it, the transport protocol handles packet loss and out-of-order delivery problems automatically. A connection-oriented server accepts an incoming connection from a client, and sends all communication across the connection. It receives requests from the client and sends replies. While a connection remains open, TCP provides all the needed reliability, retransmits lost data, verifies data that arrives without transmission errors, and reorders incoming packets as necessary. Finally, the server closes the connection after it completes the interaction. When a client sends a request, TCP either delivers it reliably or informs the client that the connection has been broken. The server handles responses in the same way.

For this stock match system server, no information loss is allowed during data communications. The loss of one stock order could invalidate the entire consequent price-

matching. Therefore, this model uses the most reliable connection-oriented (TCP) approach.

3.4 Single-process or Multi-processes Server

A multiple server uses a master server and slave servers. A single master server process begins execution initially, opens a socket at the well-known or a user-defined port, waits for the next request, and creates a slave server process to handle each request. The master server never communicates directly with a client, but passes that responsibility to a slave. Each slave process handles communication with one client. After the slave forms a response and sends it to the client, the slave exits. On the other hand, using a single-process server and asynchronous I/O also could manage multiple connections.

A single-process solution is used if the server must share or exchange data among connections. A multi-process solution is used if each slave can operate in isolation or to achieve maximal concurrency (e.g. on a multi-processor). In this model, the incoming stock orders are accepted by the slave server, and are sent to the master server. It seems more appropriate if the system uses a single-process because the server must share the incoming order information, and the updated stock asked-bid pending state varies upon the different incoming stocks. However, the maximal concurrency of the stock server is very important because in the real world there could be hundreds of incoming stock orders within one second, and a single concurrent process can not handle these incoming orders. When the single process accepts one connection and provides related service, it disregards all the remaining incoming connections. Server blocking could arise in a more

subtle way if the client does not respond to or acknowledge an incoming order. If the server process is blocked, it obviously cannot handle other connections. Server blocking is a serious liability in the server because it means the behavior of one client can prevent the server from handling other clients. This also means the entire stock order receipt system is blocked! Therefore, a single process server is not appropriate and the multi-processes approach is the only choice.

Using the IPC mechanism, the master server spawns a new slave server, the slave accepts the stock order, sends it to the master server to do the price-matching, and then exits. Several techniques could achieve this and this model uses pipes. The additional procedure the master needs to do is to set up a pipe between itself and the spawned slave before the server connection to any client. Upon a client connection request, the slave server is spawned to accept the order, then it sends the order to the master server, and exits.

3.5 Concurrent Server Algorithm

3.5.1 A Generic Concurrent, Connection-Oriented Server Algorithm

Connection-oriented application protocols use a connection as the basic paradigm for communication. They allow a client to establish a connection to a server, communicate over that connection, and then discard it. In most cases, the connection between client and server handles more than a single request. The protocol allows a client to send repeated requests and to receive responses without terminating the connection or creating a new one. Thus, connection-oriented servers implement

concurrency among connections.

Algorithm:

Master 1. **Create a socket and bind to the well-known or a user-defined address for the service being offered. Leave the socket unconnected.**

Master 2. **Place the socket in passive mode, make it ready for use by a server.**

Master 3. **Repeatedly call accept() to receive the next request from a client, and create a new slave process to handle the response.**

Slave 1. **Receive a connection request (i.e., socket for the connection) upon creation.**

Slave 2. **Interact with the client using the connection: read request(s) and send back response(s).**

Slave 3. **Close the connection and exit. The slave process exits after handling all requests from one client.**

3.5.2 Process Structure

Figure 3.1 illustrates the process structure of a concurrent, connection-oriented server. The master server process does not communicate with clients directly. Instead, it waits at the well-known or a user-defined port for the next connection request. Once a request arrives, the system returns the socket descriptor of the new socket to use for that connection. The master server process creates a slave process to handle the connection,

and allows the slave to operate concurrently. At all times, the server consists of one master process and zero or more slave processes.

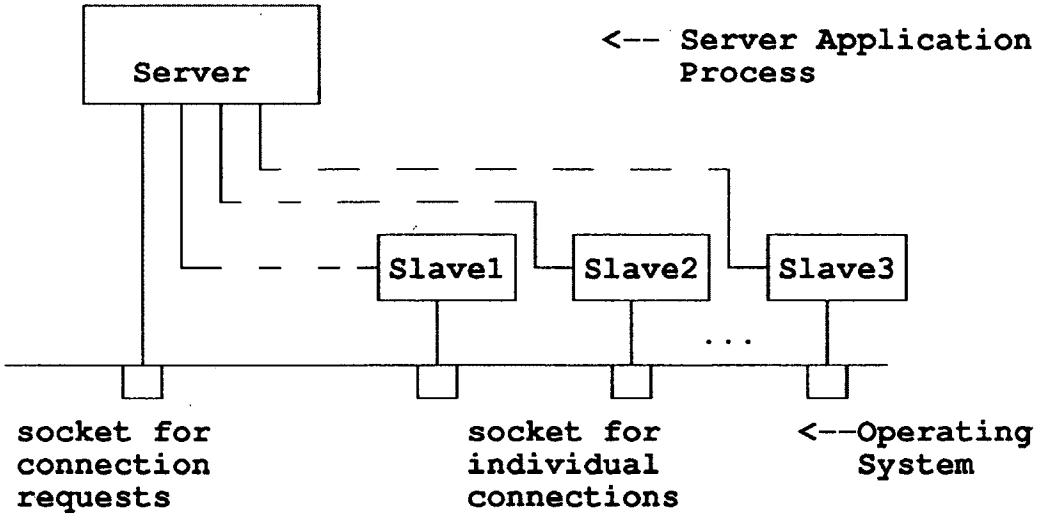


Figure 3.1 Process structure of a concurrent, connection-oriented server. A master server process accepts each incoming connection, and creates a slave process to handle it.

3.6 A Modified Concurrent Server Design and Algorithm

3.6.1 Server Service Requirement

In this model, the system server is responsible for accepting stock orders, providing stock trade information to clients, and price matching. In the implementation, the server must:

1. Accept an order, then
2. Send an order-received acknowledgement to the client,
3. Process price-matching of the stock order, then
4. Update the order data list.

5. Send an order-matched acknowledgement to the client, or

6. Accept an information request, then

7. Send the requested information to the client.

Among the server's responsibilities are the concurrency of the order receipt and the concurrency of the information service. In addition, a price-matching process is required to accomplish the price matching.

The stock order-pending state information the server provides to clients will be updated dynamically as new order comes and sometimes in deletion of the old order in the order list due to the price matching. This means that information client2 gets from the server might be different from that client1 got one minute earlier because some of the stock orders have been executed and removed from the order list.

3.6.2 System Server Structure

There are two concurrency issues in the master server: the concurrency of accepting orders and the concurrency of information service. If one master server handles both concurrences, the master server could be overloaded. Thus, the separation of the two concurrences is needed for effective client/server system performance.

Now, let us analyze the system layout again from the network point of view.

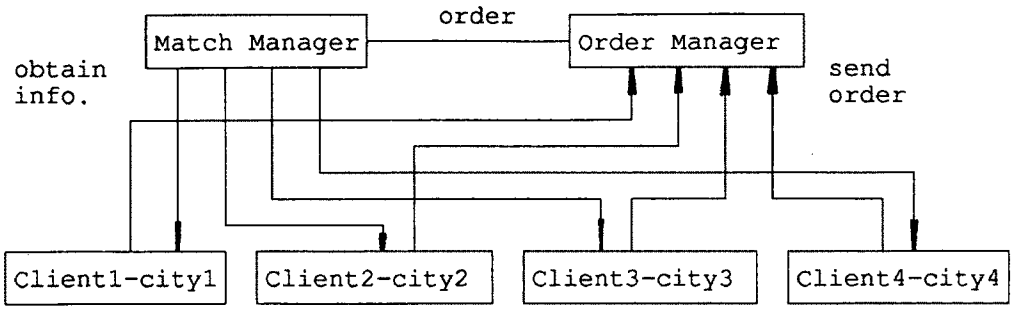


Figure 3.2 System layout from the network point of view.

The system server consists of the Match Manager and the Order Manager, which communicate by an inter-process communication mechanism. Theoretically, pipes, a shared memory with semaphore synchronization, and a message queue can be used. Comparing the pipe with the semaphore, the IPC performance of semaphore assumes that a shared memory segment is being used for the actual message storage. In this case, the time required for the system to process a message in the shared memory segment depends on the synchronization overhead, i.e., the semaphore time. An example of semaphore and shared memory IPC is shown as follows:

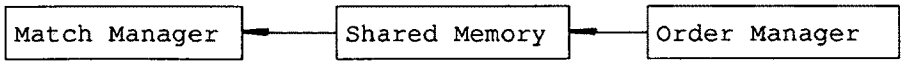


Figure 3.3 IPC layout using shared memory and semaphores

The shared memory works as a circular queue in which stock orders are stored. Using FIFO, the Order Manager receives orders from clients and puts them into the shared memory; the Match Manager keeps checking the incoming new orders and removes them from the memory while the semaphore synchronizes the operations of the two managers. While the shared memory is empty, the Match Manager can not take order from the memory segment; while the shared memory is full, the Order Manager can not put new order in the memory segment. The pipes use standard UNIX block buffers for holding each unit of data in the kernel; the time needed to transfer the message is slightly affected by the buffer size. However, semaphore with shared memory has a disadvantage. The Match Manager would go to sleep and do nothing when the shared memory is locked by the Order Manager. The Match Manager would spend much time on waiting for the memory is available. But the Match Manager has other tasks to accomplish such as the price matching and concurrent information services, so that if it spends a large amount time on retrieving orders, it will slow down the entire system performance. Using pipes, one can easily set up non-blocking pipes and embed them in the X Window System by using XtAddInput() call. This will be discussed in the next chapter.

Generally, message queue, shared memory, and semaphore are implemented to meet special IPC requirements. Message queue allows different lengths of data to pass through the queue. Semaphore is used as synchronization primitive, not useful for exchanging large amounts of data, as are pipes, FIFO and message queue. Semaphore is commonly used to synchronize access to shared memory segments. Also, processes

have to coordinate the use of the memory among themselves. If semaphores are used for the synchronization, then while one process is reading into the shared memory, other processes must wait for the read to finish before processing data. In this design, the Match Manager still needs to read into buffer while the Order Manager writes a new order into the buffer, which is exactly the way pipes works. Therefore, pipes are used in this project.

Both the Match Manager server and the Order Manager server are implemented as concurrent connection-oriented servers.

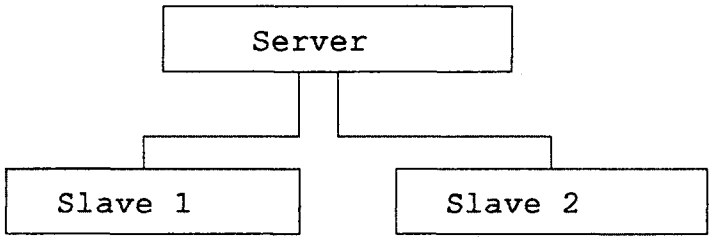


Figure 3.4 There is no intensive IPC between a typical server and its slaves.

As Figure 3.4 shows, there is no intensive IPC between the master server and its slaves after slaves are generated. Slaves inherit all the information from the master and communicate with clients independently. However, This is not the case in the implementation of the Order Manager.

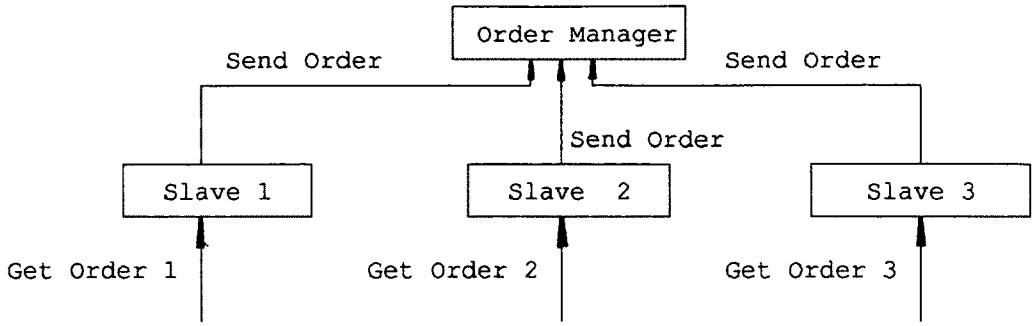


Figure 3.5 The Order Manager forks slaves to accept orders. After slaves received orders, the Order Manager import orders from slaves.

Upon the receipt of a new order, the master server forks a slave to accept the order, the slave server sends the received order to the master server by some IPC mechanism. Notice that this is different from the conventional concurrent server implementation. Among pipes, shared memory, and message queue IPC techniques, pipes are provided with all flavors of UNIX and are easy to implement. If the slave process writes less than the capacity of a pipe, the write is guaranteed to be atomic. If two slaves each writes an order message at about the same time, either all the data from the first slave is written, followed by all the data from the second slave, or vice versa. The system does not mix the data from the two slaves. This mechanism will prevent two incoming orders from being mixed up.

The pipes should be set up as O_NODELAY because the master server should not be blocked if there is no order available in the pipe. Suppose a pipe between the master and one of the slaves is malfunctioning because the slave sends messages to the master, but the master keeps reading and being blocked. The master will not be able to fork

another slave to accept other orders.

The master Order Manager server will set up a pipe between itself and its slave before any client connection is made. Then the slave, after setting up a connection to the client and accepting an order, sends an acknowledgement to the client, indicating that the order has been received successfully, and finally the slave sends the order to the master Order Manager. The master Order Manager then sends the order to the Match Manager for price matching.

Algorithm 3.1 /* Order Manager */

```
Step 1: Initialization
        Set up socket on server site ORDER_RECE_PORT (3000).
Step 2: Wait for a client's order.
        If no order comes then goto Step 2
        else Step 3.
Step 3: Generate a slave server, set up a pipe between the
        master server and the slave server.
Step 4: The slave server receives the new order and sends it
        to the master server.
Step 5: The master server receives the order and sends it to
        the Match Manager. Goto step 2.
```

The concurrency in the Match Manager is to provide stock information request concurrently. The Match Manager accepts orders from the Order Manager, processes price matching, sends order-matched acknowledgements to clients, and provides trading information. One difference between the Match Manager and the Order Manager is the I/O multiplexing. Since the Match Manager will accept inputs from both the Order

Manager and clients, the I/O multiplexing techniques is used in order to maximize the CPU utilization. Section 4.4 will discuss this in detail.

Algorithm 3.2 /* Match Manager */

```
Step 1: Initialization
        Set up socket on server site ORDER_REPORT_PORT
        (3001).
Step 2: XtMainLoop().
        Select from the following file descriptors.
        1. Pipe for new orders from the Order Manager.
        2. Socket for stock order-pending state histogram
           request from clients.
Step 4: if message arrives at file descriptor (1) then
        receive order, process price-matching, send ACK to the
        sending client.
        if message arrives at file descriptor (2) then
        fork slave server, arrange the required information
        and send it to the requesting client.
```

3.7 System Client

The system client provides the interface between brokers and the system server. The graphic user interface allows brokers to easily perform various stock transaction operations, such as delivering orders and requesting current stock order-pending state reports. Brokers could manually input the desired stock order. One could also make a simulated stock order generated by the computer at anytime. Also, the client would

receive two acknowledgements, one is the ACK for the order received, another is the confirmation to the price matching.

Algorithm 3.3 /* System Client */

```
Step 1: Initialization
        1. Setup socket on client site ORDER_RECE_PORT (3000).
        2. Setup socket on client site ORDER_REPORT_PORT
           (3001).
        3. Setup socket on server site ACK_PORT (3002).
Step 2: MOTIF GUI Setup.
        1. MenuBar.
        2. Stock List Field.
        3. Stock Information Field.
        4. Message Board.
Step 3: XtMainLoop().
        Wait for XEvent corresponding to the following events:
        1. User input stock order.
        2. Send simulated stock orders.
        3. Stop sending simulated stock orders.
        4. Request stock order-pending state report.
```

3.8 The Socket Interface

3.8.1 The Socket Call

An application calls socket() to create a new socket that can be used for network communication. The call returns a descriptor for the newly created socket. Arguments to the call specify the protocol family that the application will use (e.g., PF_INET for TCP/IP) and the protocol or type of service it needs (i.e., stream or datagram). For a socket that uses the Internet protocol family, the protocol or type of service argument determines whether the socket will use TCP or UDP. In this model, TCP protocol is used.

3.8.2 The Connection Call

After creating a socket, a client calls connect() to establish an active connection to a remote server. An argument to connect() allows the client to specify the remote endpoint, which includes the remote machine's IP address and protocol port number. Once a connection has been made, a client transfers data across it.

3.8.3 The Write Call

Both clients and servers use write() to send data across a TCP connection. Clients usually use write() to send requests, while servers use it to send replies. A call to write requires three arguments. The application passes the descriptor of a socket to which the data should be sent, the address of the data to sent, and the length of the data. Usually, write() copies outgoing data into buffers in the operating system kernel, and allows the

application to continue execution while it transmits the data across the network. If the system buffers become full, the write() may block temporarily until TCP can send data across the network and make space in the buffer for new data.

3.8.4 The Read Call

Both clients and servers use read() to receive data from a TCP connection. Usually, after a connection has been established, the server uses read() to receive a request that the client sends by calling write(). After sending its request, the client uses read() to receive a reply.

To read from a connection, an application calls read() with three arguments. The first specifies the socket descriptor to use, the second specifies the address of a buffer, and the third specifies the length of the buffer. Read() extracts data bytes that have arrived at that socket, and copies them to the user's buffer area. If no data have arrived, the call to read blocks until it does. If more data have arrived than fit into the buffer, read extracts only enough to fill the buffer. If fewer data have arrived than fit into the buffer, read extracts all the data and returns the number of bytes it found.

3.8.5 The Close Call

Once a client or server finishes using a socket, it calls close() to deallocate it. If only one process is using the socket, close() immediately terminates the connection and deallocates the socket. If several processes share a socket, close() decrements a reference count and deallocates the socket when the reference count reaches zero.

3.8.6 The Bind Call

When a socket is created, it does not have any notion of endpoint addresses (neither the local nor remote address is assigned). An application calls bind() to specify the local endpoint address for a socket. The call takes arguments that specify a socket descriptor and an endpoint address. For TCP/IP protocols, the endpoint address uses the sockaddr_in structure, which includes both an IP address and a protocol port number. Primarily, servers use bind() to specify the well-known or a user-defined port at which they will await connections.

3.8.7 The Listen Call

When a socket is created, the socket is neither active (i.e., ready for use by a client) nor passive (i.e., ready for use by a server) until the application takes further action. Connection-oriented servers call listen() to place a socket in passive mode and make it ready to accept incoming connections.

Most servers consist of an infinite loop that accepts the next incoming connection, handles it, and then returns to accept the next connection. Even if handling a given connection takes only a few milliseconds, it may happen that a new connection request arrives during the time the server is busy handling an existing request. To ensure that no connection request is lost, a server must pass listen() an argument that tells the operating system to enqueue connection requests for a socket. Thus, one argument of the listen() call specifies a socket to be placed in passive mode, while the other one specifies the size of the queue to used for that socket.

3.8.8 The Accept Call

After a server calls socket() to create a socket, bind() to specify a local endpoint address, and listen() to place it in passive mode, the server calls accept() to extract the next incoming connection request. An argument to accept() specifies the socket from which a connection should be accepted.

Accept() creates a new socket for each new connection request, and returns the descriptor of the new socket to its caller. The server uses the new socket only for the new connection; it uses the original socket to accept additional connection requests. Once it has accepted a connection, the server can transfer data on the new socket. After it finishes using the new socket, the server closes it.

3.8.9 Using Socket Calls in a Program

The following figure illustrates a sequence of calls make by a client and a server using TCP.

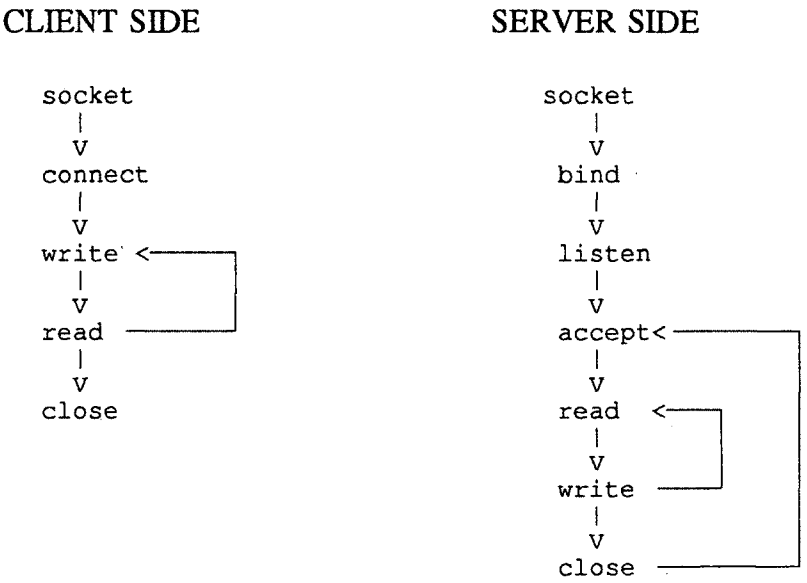


Figure 3.6 An example sequence of socket system calls made by a simple client and a server using TCP. The server runs in a loop, waiting for new connections on the user-defined port. It accepts clients' requests by setting up new connections, provides services, then closes the connections.

The client creates a socket, calls connection() to connect to the server, and then interacts using write() to send requests and read() to receive replies. When it finishes using the connection(), it calls close(). A server uses bind() to specify the local (well-known or a user-defined) protocol port it will use, calls listen() to set the length of the connection queue, and then enters a loop. Inside the loop, the server calls accept() to wait until the next connection request arrives, uses read() and write() to interact with the client, and finally uses close() to terminate the connection. The server then returns to the accept() call, where it waits for the next connection.

Chapter 4 CLIENT SYSTEM USER INTERFACE

4.1 Why MOTIF?

The system is built using the OSF/MOTIF GUI because MOTIF implementation gains immediate compatibility with a large body of existing UNIX applications based on X Window System communications and display protocol. The underlying issue for this project is how to embed IPC (socket) mechanism into the MOTIF environment.

4.2 Motif Widget Overview

The OSF/MOTIF widget set is based on the Xt Intrinsics, a set of functions and procedures that provide quick and easy access to the lower levels of the X Window system. Figure 4.1 shows that the Motif Widget system is layered on top of the Xt Intrinsics, which in turn are layered on top of the X Window System, thus extending the basic abstractions provided by X.

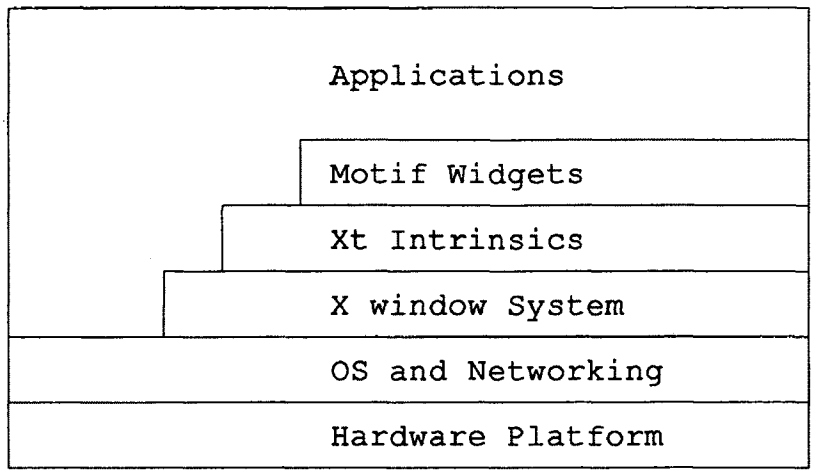


Figure 4.1 Programmers' view of the X Window System with a user interface model.

The Motif Widget system supports independent development of new or extended widgets. The Motif Widget system consists of a number of different widgets, each of which can be used independently or in combination to aid in creating complex applications. One can write applications faster and with fewer lines of code using the Motif Widgets. However, Motif Widgets require more memory than similar applications written without them.

The basic steps in writing widget programs are:

1. Include required header files	#include <X11/Intrinsic.h>
	#include <Xm/Xm.h>
	#include <Xm/*widget*.h>
2. Initialize Xt Intrinsics	XtInitialize()
3. Add additional top-level windows	XtAppCreateShell()

4. Do steps 4 through 6 for each widget,

Set up argument lists for widget	XtSetArg()
5. Add Callback routines	XtAddCallback()
6. Create widget	XtCreateManagedWidget()
	or XmCreate*widget*
	followed by
	XtManageChild(*widget*)
7. Realize widgets and loop	XtRealizeWidget(*parent*)
	XtMainLoop()
8. Link widget library	
9. Create defaults files	

Every widget is dynamically allocated and contains state information. Every widget belongs to one class, and each class has a structure that is statically allocated and initialized and contains operations for that class. Widgets are used either individually or in combination to make the creation of complex application easier and faster. Some widgets display information, others are merely containers for other widgets. Some widgets are restricted to displaying information and do not react to keyboard or mouse input, others change their display in response to input and can invoke functions when instructed to do so. One can customize some aspects of a widget, such as fonts, foreground and background colors, border widths and colors, and sizes.

4.3 User Interface Design

The design of client GUI follows the general MOTIF programming rules and its style. The widget hierarchy of the client GUI is shown as follows:

```
* Top-Level Shell
  * Main Window Widget  ( main )
    * Menubar Widget    ( menubar )
      * Cascade Button  ( file menu )
      * Pulldown Menu   ( file )
          * Pushbutton Widget  ( "Get File Info" choice )
          * Pushbutton Widget  ( "Exit" choice )
      * Cascade Button        ( edit menu )
        * Pulldown Menu         ( edit menu )
          * Pushbutton Widget  ( "Undo" choice )
          * Separator  Widget
          * Pushbutton Widget  ( "Cut" choice )
          * Pushbutton Widget  ( "Copy" choice )
          * Pushbutton Widget  ( "Paste" choice )
      * Cascade Button        ( option menu )
        * Pulldown Menu       ( option menu )
          * Pushbutton Widget  ( "Order" choice )
          * Pushbutton Widget  ( "Autosend" choice )
          * Pushbutton Widget  ( "Pause autosend" choice )
          * Pushbutton Widget  ( "Pending state" choice )
    * FormWidget              ( form widget )
```

```
* RowColumnWidget    ( rowcolumn )
  * ScrolledList     ( list-widget )
    * FormDialogWidget     ( graph-form )
      * MainWindowWidget   ( st_gre_main )
        * MenuBarWidget    ( main_menubar )
        * DrawingAreaWidget ( graph_area )
          * LabelWidget      ( label )
* ScrolledWindowWidget ( "indicator_win" )
  * RowColumnWidget ( "Market Indicator" )
    * LabelWidget
    * LabelWidget
    .
    .
    .
* TextWidget        ( "Message Board" )
```

All the widgets under the Top-Level Shell are the subclasses of it. The first level below the Top-Level Shell is the Main Window Widget, which provides a standard layout for the primary window of an application. This layout includes a MenuBar, a CommandWindow, a work region, and a ScrollBar. All of these areas are optional. In a fully-loaded MainWindow, the MenuBar spans the top of the window horizontally. The CommandWindow spans the MainWindow horizontally just below the MenuBar, and the work region lies below the CommandWindow. For the GUI of the client system, the MenuBar and the work region, provided under the MenuBarWidget and the FormWidget in the above hierarchy, virtually consist of the stock transaction commands, the stock list

area, the stock information field, and the message board.

Under the FormWidget, there are three child widgets: 1. ScrolledListWidget 2: RowColumnWidget, 3. TextWidget. The ScrolledListWidget has the function of presenting items in a list and letting the user choose items from the list. The optional scroll bars allows the user to scroll through the list area. An instance of ListWidget can be created by the following program:

```
#include <Xm/List.h>
listwidget = XmCreateList(parent,name,NULL,0);
```

The single selection policy allows user select one of the stock items.

```
Arg  args[10];
int  n;
n = 0;
XtSetArg(args[n],XmNselectionPolicy, XmSINGLE_SELECT); n++;
/* create list widget */
```

When one of the stock items is selected, a FormDialogWidget will pop up with another MainWindowWidget attached to it. This MainWindowWidget consists of a MenuBar, a DrawingAreaWidget where the stock order pending state will be displayed, a vertical scroll bar, and a horizontal scroll bar. The following codes show the creation of a FormDialogWidget:

```
#include  <Xm/Form.h>
Widget    parent;
String    name;
widget = XmCreateFormDialog(parent, name, NULL, 0);
```

The optional scroll bar allows the user to move around part of the pending state

histogram. The way to create an instance of a DrawingAreaWidget is similar to that of ListWidget except the function name is different. By setting the desired widget resources, one could get a good-looking layout for each widget. The widget resources could be set either before the creation of a widget by

```
int n;
Arg  args[10];
n = 0;
XtSetArg(args[n],XmNx,300); n++;
XtSetArg(args[n],XmNy,700); n++;
```

or after the creation of a widget by

```
int n;
Arg  args[10];
n = 0;
XtSetArg(args[n],XmNx,300); n++;
XtSetArg(args[n],XmNy,700); n++;
XtSetArgValues(widget,args,n);
```

The DrawingAreaWidget provides a blank rectangular drawing area that a program can do with as it pleases by using low-level X library calls. In this implementation, it is used to display the stock asked-bid order-pending state histogram. Two common techniques can implement the display.

1. Using low level X library calls such as XFillRectangle(), XSetForeground(). Although the drawing speed is fast, whenever one wants to attach some text on each rectangle, more complicated procedures are needed in order to get the window position in which the text should be located. In a situation that there are hundreds of rectangles

with texts, the total number of additional procedures will be very large.

2. Representing rectangle in terms of LabelWidget. It is easier to control the position of the rectangle by setting the XmNx, XmNy resources. The XmNlabelString resource makes it possible to attach a text string on the rectangle regardless of the position of the rectangle. Another advantage is some mouse-pressed callbacks allow the end user to access the additional information such as stock's volume by clicking the label (rectangle.)

The "Option" choice under the MainWindow menubar allows brokers to display the stock pending state histogram by manually inputing the preferred stock name. This feature is added in case that there are too many stock items under the scrolled list. Otherwise, it might be inconvenient for brokers to move the scroll bar and follow the moving list to catch the stock item. Three other main operation commands under the "Option" pulldown menu allow brokers to: 1. make a stock order, 2. automatically send simulated stock, 3. pause the simulated stock sending. A BulletinBoardDialogWidget will pop up to let brokers enter the desired stock and related order information. The following shows the widget tree of the BulletinBoardDialogWidget:

```
* ScrolledList          ( ListWidget )
  * BulletinBoardDialog  ( BulletinBoardDialog )
    * PushButtonWidget     ( "StockName" )
    * PushButtonWidget     ( "Volume" )
    * PushButtonWidget     ( "Price" )
    * PushButtonWidget     ( "Type" )
    * PushButtonWidget     ( "NoneOrAll" )
```

```
* TextWidget          ( "Field 1" )
* TextWidget          ( "Field 2" )
* TextWidget          ( "Field 3" )
* TextWidget          ( "Field 4" )
* TextWidget          ( "Field 5" )
* PushButtonWidget    ( "SendOrder" )
```

The first five PushBottonWidgets label the information name and prompt to brokers to enter the information in the corresponding TextWidget. The "SendOrder" PushButtonWidget will format the order information and send it the system server. The order structure combines all the order information with the client's IP address, which will be referred when the server sends the order-received ACK and the order-matched ACK to the client.

The order information can be retrieved by calling XmTextGetString(textwidget). The following shows the creation of the BulletinBoardWidget:

```
int     i,n;
Arg     args[10];
n=0;
XtSetArg(args[n],XmNautoUnmanage,FALSE); n++;
perorder_bulletin=XmCreateBulletinBoardDialog(parent,
                                name,
                                args,
                                n);
for ( i=0; i<XtNumber(stock_ele);i++){
        element[i]=XtCreateWidget(stock_ele[i],
                                xmtextwidgetclass,
```

```
                                perorder_bulletin,
                                NULL,
                                0);
        }
        for ( i=0; i<XtNumber(buttons); i++) {
                button[i]=XtCreateWidget(buttons[i],
                                xmpushbuttonwidgetclass,
                                perorder_bulletin,
                                NULL,
                                0);
        }
        send_button=XtCreateManagedWidget("send_order",
                                xmpushbuttonwidgetclass,
                                perorder_bulletin,
                                NULL,
                                0);
        send_button=XtCreateManagedWidget("send_order",
                                xmpushbuttonwidgetclass,
                                perorder_bulletin,
                                NULL,
                                0);
        XtAddCallback(send_button,XmNactivateCallback,
                        deliverorder_callback,element);
        XtManageChildren(element,XtNumber(stock_ele));
        XtManageChildren(button,XtNumber(buttons));
}
```

4.4 Handling Other I/O Sources

As mentioned in chapter 2, the Match Manager process reads input from more than one source (multiplexing I/O): 1. requests from clients for sending trading information, and 2. orders sent from the Order Manager. In the first case, the process opens a socket to receive a request from a client. In the second case, a pipe should be set up between the Match Manager and the Order Manager. There are a few different techniques available to handle the multiplexing of different I/O channels.

1. One can set a socket to nonblocking, using either the FNDELAY flag to fcntl() or the FIONBIO request to ioctl(). But the process has to execute a loop that reads from the socket and pipe, and if nothing is available to read, wait some amount of time before trying the read again. This is called *polling*. The software polls each socket at some interval (every second, perhaps) and if no data are available to read, the process waits by calling the sleep() function. Polling can waste computer resources because most of the time there is no work to be done. Obviously, this technique is not appropriate for the implementation of the Match Manager process because it has another important task:price matching. Leaving a large amount of time to price-matching processing is critical to the entire system performance.

2. The process can fork one child process to handle each I/O channel. In this example, it spawns two processes, one to read from the socket and one to read from the pipe. Each child process calls read() and blocks until data are available. When a child returns from a read, it passes the data to the parent process using some form of IPC (which the parent must set up before spawning its children). Any techniques:pipe, FIFO,

message queue, shared memory with a semaphore, or another socket also can be used.

3. Asynchronous I/O can be used. The problem with this method is that signals are expensive to catch. Also, if more than one descriptor has been enabled for asynchronous I/O, the occurrence of the SIGIO signal doesn't tell which descriptor is ready for I/O. This method available only for terminals and sockets under 4.3 BSD.

4. Use alarm to get a SIGALRM every second and check the input socket for data. Thus the XNextEvent loop works normally except it gets alarmed every second and data get plotted from the signal handler. Although XNextEvent will get interrupted by signals such as SIGALRM, since the X libraries are not reentrant, the results are unpredictable.

5. Another technique is provided by 4.3BSD with its select() system call. This system call allows the user process to instruct the kernel to wait for any one of multiple events to occur and to wake up the process only when one of these events occurs. The select() call could take either the X connection descriptor or any other socket or pipes descriptor in which we are interested. The XtAppAddInput() would do all the descriptor selecting.

CHAPTER 5 PRICE MATCHING

5.1 Order's Data Structure

There are three common types of orders: 1. *market price order*, 2. *limit order to buy*, and 3. *limit order to sell* in the stock market. Orders also could specify extra requirements such as "none or all". For each stock, two waiting lists are set up for bid orders and asked orders. Each order is inserted in a waiting list according to the stock transaction rules.

1. **Market price order has higher priority than other types of orders.**

The next two rules determine the relative priority of two market price orders.

2. **A market price order with a smaller volume has higher priority than a market order with larger volume.**
3. **For market price orders with the same volume, the market order which is already in the list has higher priority.**

The next three rules determine the relative priority of two non market price orders.

4. **Within all bid orders except market price orders, the order with highest price gets priority.**
5. **Within all asked orders except market price orders, the order with lowest price gets priority.**
6. **For two orders with the same price, the order which is already in the list has higher priority.**

All stock orders except the market price order are in ascendant order sorted by

order's price. Because of the special attribute of the market price order, (asking for the best buy or sell price available in the current market) it is inserted into the waiting list in descent by order's volume in bid order list and in ascent by order's volume in asked order list. The Match Manager will create a list for each bid order queue and asked order queue. Assume the list for bid order is B(m), the list for asked order is A(n), where m, n is the order index. A market price order is placed differently from the bid order list and the asked order list.

Example 1:

If B(1), B(2), B(3) ... B(m) are all bid orders (except the market price order), according to the stock market rules, the first market price order will be inserted at the end of the list, so the new list is B(1), B(2), B(3) ... B(m), B(m+1), where B(m+1) is a market price order. Next, another market price order is inserted. Depending on the volume of the new order, it could be inserted either between B(m) and B(m+1) if the new order's volume is greater than or equal to the B(m+1).volume, or at the end of the list if its volume is less than the B(m+1).volume. This policy will give the market order the maximum possibility of being matched.

Example 2:

If A(1), A(2), A(3) ... A(n) are all asked orders (except the market price order), according to the stock market rules, the first market price order A_m(1) will be inserted at the first of the list, so the new list is A_m(1), A((1), A(2), A(3) ... A(m). Next, another market price order is inserted. Depending on the volume of the new order, it could be inserted either between A_m(1) and A(1) if the new order's volume is greater

than or equal to the A_m(1).volume, or at the first of the list if its volume is less than A_m(1).volume. Again, this will give the market price order the maximum possibility of being matched.

Algorithm 5.1 **/* for bid order list insertion */**

```
If order N is market price order then
  if there are market price orders in the list then find the
      first market price order M(k) order in the list,
      compare N.volume to M(k).volume.
      1: if M(k).volume <= N.volume then insert N between
         M(k-1) and M(k).
         else find the second market price order M(k+1)
         goto 1.
  else insert the order at the end of the list.
else insert N by price in ascendant order.
```

Algorithm 5.2 **/* for asked order list insertion */**

```
If order N is market price order then
   if the first order in the list is market price order then
      compare N.volume to M(1).volume.
      1: if M(1).volume <= N.volume then insert N between
      M(1) and M(2).
      else find the second market price order M(2) goto 1.
   else insert N at first of the list.
else find first non-market price order in the list. insert
     N by price in ascendant order.
```

The logical allocation of orders during the implementation will be a linked list instead of an array because:

1. The number of orders in the memory is not static but dynamic.
2. The price matching processing requires extensive insertion and deletion operations, which are not suitable to the array memory allocation mechanism.

5.2 Price Matching Algorithm

Based on the stock market rules, the price match algorithm is designed to match the price of different types of orders. The price matching proceeds within one stock with the input of the corresponding stock bid order list and asked order list. Assume cur_bid is the current bid order that will be matched in the bid order list, and cur_ask is the current asked order that will be matched in the asked order list. At the beginning of the price matching, the cur_bid is the last order of the bid order list, and the cur_ask is the first order of the asked order list. The previous order of cur_bid is the order immediately before cur_bid in the bid order list, and order next to cur_ask is the order immediately next to cur_ask in the asked order list. Both the previous order of cur_bid and order next to cur_ask have the second highest priority of being matched.

The price matching is **possible** (**not always**) if:

1. One of cur_bid and cur_ask is market price order, or
2. The price of cur_bid is higher than or equal to that of cur_ask.
3. However, price matching might not succeed if the "none_or_all" of a stock order is TRUE.

For simplicity, the following assumptions for the order price matching and the order selection of the price matching are TRUE:

1. If a stock order is none_or_all, it means that one could only buy or sell the stock with its whole volume at most one time.
2. If the cur_bid is market order, and if it is impossible to match with cur_ask, then the order next to the cur_ask is selected to match with cur_bid. If the

order next to cur_ask is empty, then the previous order of cur_bid is selected to match with the first order in the asked order list until it is empty.

3. If cur_ask is market order, and if it is impossible to match with cur_bid, then the previous order of cur_bid is selected to match with cur_ask. If the previous order of cur_bid is empty, then the order next to cur_ask is selected to match the last order in the bid order list until it is empty.
4. If neither cur_ask nor cur_bid is market order and if cur_bid is impossible to match with cur_ask, then the order next to cur_ask is selected to match with cur_bid. If the order next to cur_ask is empty, then the previous order cur_bid is selected to match with the first order of the asked order list until it is empty.

Algorithm 5.3 **/* for price matching */**

```
price_match()
input: B(1),B(2),B(3) ... B(m), cur_bid; /* bid order list and
       current match bid order */
       A(1),A(2),A(3) ... A(n), cur_ask; /* asked order list
       and current match asked order */
{
    cur_bid = B(m); cur_ask = A(1);
    if cur_bid is market price order then bid_market_match()
    else
    if cur_ask is market price order then ask_market_match();
    else regular_match();
}
bid_market_match()
```

```
input: B(1),B(2),B(3), ... B(m), cur_bid; /* bid order list
       and current bid match order; */
       A(1),A(2),A(3), ... A(n), cur_ask; /* asked order list
       and current asked match order; */
{
    if cur_bid.volume > cur_ask.volume then
       if cur_bid.none_or_all is FALSE then
          cur_bid.volume = cur_bid.volume - cur_ask.volume;
          delete(cur_ask);
          send price match ACK to the sender of cur_ask;
          bid_market_match();
      else
          let cur_ask = next cur_ask;
          if ( cur_ask != NULL ) price_match();
          else
             let cur_bid = previous cur_bid;
             if ( cur_bid != NULL ) price_match();
             else return;
    else
       if cur_bid.volume = cur_ask.volume
          delete cur_bid, cur_ask;
          send order-matched ACK to clients;
          price_match();
    else
       if cur_ask.none_or_all is FALSE then
          cur_ask.volume = cur_ask.volume - cur_bid.volume;
```

```
            delete(cur_bid);
            send price match ACK to the send of cur_bid;
            price_match();
         else
            let cur_ask = next cur_ask;
            if ( cur_ask != NULL ) price_match();
            else
               let cur_bid = previous cur_bid;
               if ( cur_bid != NULL ) price_match();
               else return;
}
ask_market_match()
input: B(1),B(2),B(3), ... B(m), cur_bid; /* bid order list
       and current bid match order;
       A(1),A(2),A(3), ... A(n), cur_ask; /* asked order list
       and current asked match order;
{
    if cur_ask.volume >= cur_bid.volume then
       if cur_ask.none_or_all is FALSE then
          cur_ask.volume = cur_ask.volume - cur_bid.volume;
          delete(cur_bid);
          send price match ACK to the sender of cur_bid;
          ask_price_match();
      else
          let cur_bid = previous cur_bid;
          if ( cur_bid != NULL ) price_match();
```

```
            else
               let cur_ask = next cur_ask;
               if ( cur_ask != NULL ) price_match();
               else return;
      else
         if cur_bid.volume = cur_ask.volume
            delete cur_bid, cur_ask;
            send order-matched ACK to clients;
            price_match();
      else
         if cur_bid.none_or_all is FALSE then
            cur_bid.volume = cur_bid.volume - cur_ask.volume;
            delete(cur_ask);
            send price match ACK to the send of cur_ask;
            price_match();
         else
            let cur_bid = previous cur_bid;
            if ( cur_bid != NULL ) price_match();
            else
               let cur_ask = next cur_ask;
               if ( cur_ask != NULL ) price_match();
               else return;
}
regular_match()
input: B(1),B(2),B(3), ... B(m), cur_bid; /* bid order list
       and current bid match order;
```

```
    A(1),A(2),A(3), ... A(n), cur_ask; /* asked order list
    and current asked match order;
{
  if cur_bid.price >= cur_ask.price then
    if cur_bid.volume < cur_ask.volume then
      if cur_ask.none_or_all = FALSE then
        cur_ask.volume = cur_ask.volume - cur_bid.volume;
        delete(cur_bid);
        send price match ACK to the sender of cur_bid;
        price_match();
      else
        let cur_ask = next ask order;
        if ( cur_ask != NULL ) price_match();
        else
            let cur_bid = previous cur_bid;
            if ( cur_bid != NULL ) price_match();
            else return;
    else
    if cur_bid.volume = cur_ask.volume then
          delete(cur_bid);
          delete(cur_ask);
          send price match ACK to the sender of cur_bid and
          cur_ask;
          price_match();
    else
    if cur_bid.volume > cur_ask.volume then
```

```
            if cur_bid.none_or_all = FALSE then
               cur_bid.volume = cur_bid.volume - cur_ask.volume;
               delete(cur_ask);
               send price match ACK to the sender of cur_ask;
               price_match();
            else
               let cur_ask = next ask order;
               if ( cur_ask != NULL ) price_match();
               else
                   let cur_bid = previous cur_bid;
                   if ( cur_bid != NULL ) price_match();
                   else return;
      else
         let cur_bid = previous cur_bid;
         if ( cur_bid != NULL ) price_match();
         else
            let cur_ask = next cur_ask;
            if ( cur_ask != NULL ) price_match();
            else return;
}
```

The algorithm uses intensive recursive calls of price_match(), bid_price_match(), and ask_price_match(), depending on whether the order is market price order or not. Naturally, the last order in the bid order list, which could be a market price order, will match the first order in the asked order list. This section describes four typical scenarios in which the server may follow the algorithm to process the price matching.

1. The current bid order is a market price order;

2. The current asked order is a market price order;

3. The current bid order and asked order are both market price orders;

4. Neither the current bid order nor the asked order is market price order;

Let us discuss each scenario in detail.

Scenario 1: When the bid order cur_bid is a market price order and

(1). if cur_bid.volume is greater than cur_ask.volume, there are two possible situations depending on the "none_or_all" values of the cur_bid order: 1. If cur_bid.none_or_all is FALSE, i.e., this order could be bought or sold by two or more separate deals, then get the remaining volume of the cur_bid; delete the cur_ask; send order-matched ACK to the client who sends this cur_ask. Then, recursively call the bid_market_match() to match the remaining volume of this order. 2. If cur_bid.none_or_all is TRUE, i.e., this order could be only bought or sold with its whole volume size at most one time. Since the cur_bid and the cur_ask can not be matched, according to order selection assumptions, the server finds the order next to the cur_ask and recursively calls price_match(), (notice that the new cur_ask could be either a market order or a regular order). When the server reaches the end of the asked order list, it comes back and finds the previous cur_bid order, then starts again to match it with the first order in the asked list.

(2). if cur_ask.volume is equal to cur_bid.volume, then send order-matched ACK to senders; delete the cur_bid and the cur_ask, then price_match().

(3). if cur_bid.volume is less than cur_ask.volume, again there are two situations: 1. If cur_ask.none_or_all is FALSE, then get the remaining volume of the cur_ask; delete the

cur_bid; send order-matched ACK to the client who sends this cur_bid. Then the server recursively calls the price_match() to match the next bid order. This time the new order could be a market price order or a regular order, so the regular_match(), the ask_price_match(), or the bid_price_match() could be called depending on the different attributes of the bid order and the asked order. 2. If cur_ask.none_or_all is TRUE, the server finds the next order in the asked list, then calls the price_match() recursively. If the server reaches the end of the asked order list, it finds the previous cur_bid order, then starts over again with the cur_bid and first one in the asked list.

Scenario 2: When the asked order ask_bid is a market price order and

(1). if cur_ask.volume is greater than cur_bid.volume, there are two possible situations depending on the requirement of the cur_ask order: 1. If cur_ask.none_or_all is FALSE, then get the remaining volume of the cur_ask; delete the cur_bid; send order-matched ACK to the client who sends this cur_bid. Then, recursively call the ask_market_match() to match the remaining volume of this order. 2. If cur_ask.none_or_all is TRUE, this cur_ask can not be matched. So, the server finds the previous cur_bid order, then recursively calls the price_match(). When the server reaches the start of the bid order list, it tries the order next to the cur_ask with the order at the end of the bid list, recursively.

(2). if cur_ask.volume is equal to cur_bid.volume, then send order-matched ACK to senders; delete cur_bid and cur_ask, then price_match().

(3). If cur_ask.volume is less than cur_bid.volume, again there are two situations: 1. If cur_bid.none_or_all is FALSE, then get the remaining volume of the cur_bid; delete the cur_ask; send order-matched ACK to the client who sends this cur_ask. Then recursively

call the price_match() to match the next bid order. 2. If cur_bid.none_or_all is TRUE, then the server finds the previous order in the bid list, then recursively calls the price_match(). When the server reaches the start of the bid list, it comes back and tries the order next to the cur_ask with the order in end of the bid list.

Scenario 3: If both the cur_bid and the cur_ask are market price, there is not much difference from the scenario 1 or 2. Simply follows scenario 1 or 2.

Scenario 4: If neither the cur_bid nor the cur_ask is market price order, then the regular_price_match() is used. If cur_bid.price is greater than or equal to cur_ask.price, then the price matching is possible. 1. if cur_bid.volume is less than cur_ask.volume then check the attribute of cur_ask.none_or_all as same as that in scenario 1 or 2. 2. if cur_bid.volume is equal to cur_ask.volume then delete the cur_ask and the cur_bid, send order-matched ACK to both senders. 3. if cur_bid.volume is greater than cur_ask.volume then check the attribute of cur_bid.none_or_all as same as that in scenario 1 or 2.

The following example shows the price-matching sequences for scenario 2, and assumes "none_or_all" attributes of all orders are FALSE.

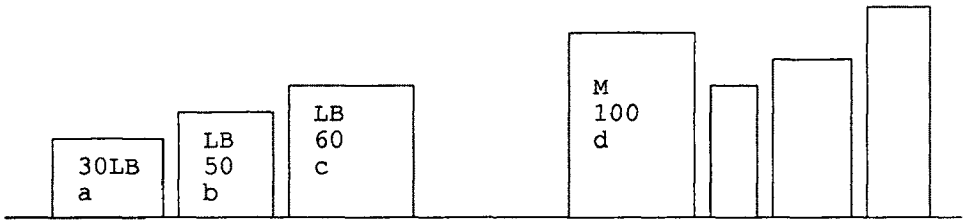


Figure 5.1 A sample asked-bid order-pending state histogram.

Figure 5.1 shows the current order list. since

Order_d.volume > Order_c.volume

the remaining Order_d.volume = 100 - 60 = 40. The matching result is shown as follows:

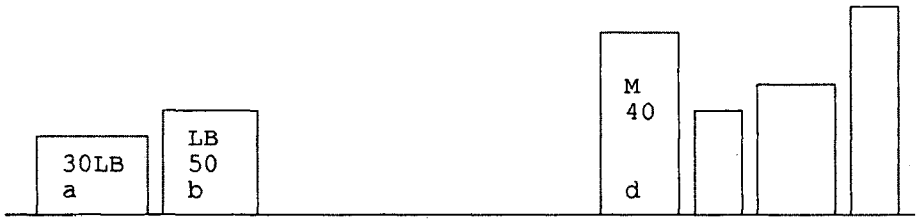


Figure 5.2 The histogram after first around price matching from Figure 5.1.

Next, notice that Order_b.volume > Order_d.volume. Therefore, this time the price matching for the market order d is completed. The remaining volume of order b is:

Order_b.volume = Order_b.volume - Order_d.volume = 50 - 40 = 10.

The result of price matching is shown as follows:

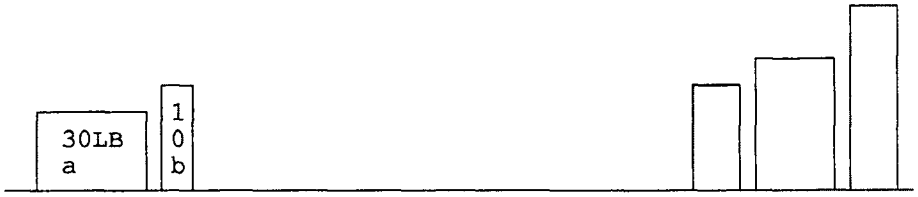


Figure 5.3 The final order-pending state after the price matching

Now, if b.none_or_all is TRUE, the result of the price match will be different, starting from the second time of price matching.

Figure 5.4 The pending-state after the first around of price matching if b.none_or_all is TRUE.

Since this time b.none_or_all is TRUE, price matching between the order b and d is impossible. The server searches for the previous available bid order a; and order a.none_or_all are FALSE. So the result would be:

Figure 5.5 The final order-pending state if b.none_or_all is TRUE.

CHAPTER 6 CONCLUSION

The system is built based upon a client/server network model by using TCP/IP protocol and its API. To design and implement the production version of the system would be very expensive since it involves both the wide and metropolitan area network management and large cumulative data storage requirements. However, the availability of such a networked information system would dramatically enhance the accuracy and efficiency of stock trading and reduce the related human cost of the floor trading in the stock markets.

This model could assist other researchers in further development of a more efficient stock trading system.

APPENDIX: Stock Technical Term:

1. Customer Account Management Panel:

A useful tool that makes it easy for brokers to manage account portfolio of their customers. The order history of their customers can be saved in time and retrieved in the future.

2. Market Price Order:

A type of stock orders that asking the broker for the best buy or sell price available in the current market.

3. Limit Order to Buy:

A type of stock orders that asking the broker not buy the stock until he could do better than the price the client desires.

4. Limit Order to Sell:

A type of stock orders that asking the broker not sell the stock until he could do better than the price the client desires.

5. None Or All:

The extra requirement of a stock order that only can be sell or buy with its whole volume at most time.

REFERENCES:

[1] Douglas E. Comer, (1993), "Internetworking with TCP/IP III", pp 49-51, pp 51, pp 96-97, pp 106-108, pp, 110, pp 112, pp133.

[2] OSF, (1990), "Programmer's Guide", pp 1-2, pp 3-1,3-2.

[3] OSF, (1990), "Programmer's Reference", pp 1-410, 1-566.

[4] Douglas A. Young, (1990), "The X Window System Programming Application with Xt OSF/MOTIF".

[5] Eric F. John, (1990), "Power Programming MOTIF".

[6] W. Richard Stevens, (1990), "UNIX Network Programming", pp 110.

[7] W. Richard Stevens, (1990), "UNIX Network Programming", pp 329.

[8] W. Richard Stevens, (1990), "UNIX Network Programming", pp 682.

[9] Richard Saul Wurman, (1990), "The Wall Street Journal". pp 12-13.